第11回

論理回路のプロパティ検証技術(3)

——プロパティ検証の実際

藤田昌宏

リファレンスとなるプロパティ(仕様)を正しく表現することは，実はそれほど容易ではない．今回は，誤ったプロパティの例を示しながら，仕様のあいまいさの問題について説明する．また，時相論理式から状態遷移表現を生成するフリーのツールを紹介する．時相論理式を自分で作成し，対応する状態遷移表現(Verilog HDL記述)を生成して，それを調べるという作業を何度か繰り返すと，時相論理でかなり自由に仕様が書けるようになるという．(編集部)

前回(本誌2004年10月号，pp.136-141)までに，プロパティ検証(property checking)の基礎として，プロパティの表現法やその状態遷移表現への変換について考え，また，検証の例も示しました．その中で，プロパティ検証では，検証しようとしているプロパティそのものを正しく表現することが，実はそれほど簡単ではないということについて説明しました．

今回は，このプロパティを正しく表現するために注意するべきことや，利用できる関連技術について説明していきたいと思います．とくに，時相論理でプロパティを記述するうえで重要な概念であるeventualityについて，詳しく説明していきます．

● 記述したプロパティの正しさをどう確認するのか？

まず，前回の終盤の説明について復習したいと思います．いま，1ビット・カウンタとして，入力が2回‘1’になると，出力が1回‘1’になるという仕様のカウンタ(ただし，後述するようにこの仕様そのものが「あいまい」なのだが…)を

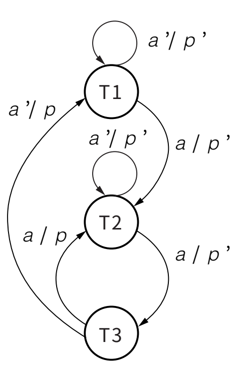


図1 1ビット・カウンタの設計例

入力信号aが2回‘1’になると，出力信号pが‘1’になる．内部でフリップフロップを一つ使って信号aが‘1’になった回数を数えるということで，ここでは1ビット・カウンタと呼ぶ．

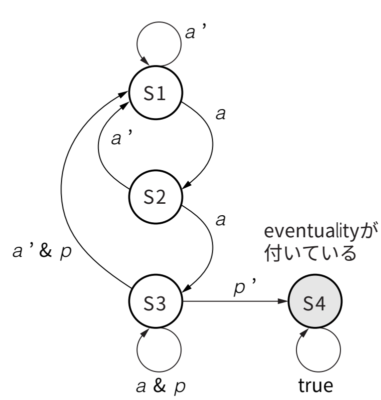


図2 プロパティ $G(a \,\&\, Xa \rightarrow XXp)$ を状態遷移に展開

1ビット・カウンタの設計に対する仕様として，「今，信号aが‘1’で，かつ，次のクロックでも‘1’であると，その次のクロックで信号pが‘1’になる」を考える．時相論理の記述は $G(a \,\&\, Xa \rightarrow XXp)$ となる．また，それを状態遷移に展開すると図のようになる．

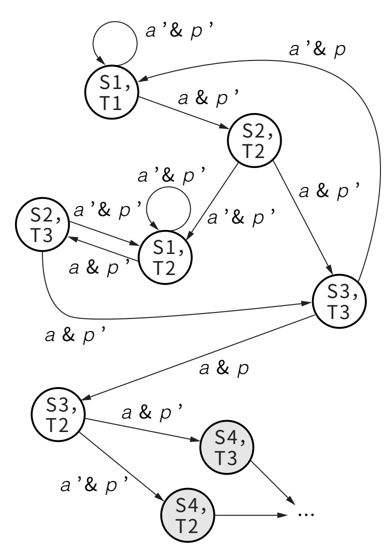


図3 図1とプロパティ $G(a \,\&\, Xa \rightarrow XXp)$ の否定(図2)の積に対応する状態遷移図

この図には状態S4が現れている．図1の設計は，1ビット・カウンタとして正しいように見えるが，実際にはプロパティを満たしていないことになる．実はこれは，カウンタの動作の詳細をどう考えるかという問題であり，検証に利用しているプロパティがかならずしもそれに沿ったものではないことが原因である．

設計することを考えます．この仕様に対する設計例を**図1**に示します．

図1の動作を初期状態から追ってみると，設計に誤りはないように思えます．合計2回，入力aの値が‘1’になる（**図1**ではaで表現している．aが‘0’であることは，$\bar{a}$で表現している）と，出力pが‘1’になっています．これに対して，プロパティとして，前回と同じように時相論理式（$G(a \,\&\, Xa \rightarrow XXp)$）を考えます．これは，信号$a$が2回連続して‘1’になると，その次の時刻に信号$p$が‘1’にならなければならないということを示しており，満たされなければならないプロパティであると言えます．さて，検証には，まずこの式の否定，

$$(G(a \,\&\, Xa \rightarrow XXp))' = F(a \,\&\, Xa \rightarrow XXp)'$$

を計算し，それを状態遷移に展開します．すると**図2**が得られます．そして，**図1**と**図2**の状態遷移の積を計算して，**図3**が得られます．**図2**において状態S4に到達すると，検証しているプロパティの否定が満たされたことになります．

状態S4にはeventualityが付いており，時相論理式を満たすには，かならずこの状態にたどり着く必要があることを示しています．このeventualityについては，後で詳しく説明します．

さて，**図3**を見てください．**図1**の設計が正しい場合，**図2**が満たされてはならないので，状態S4は結果の状態遷移図である**図3**には決して現れないはずです．しかし，現実には現れています．つまり，仕様の否定が満たされているということであり，設計は仕様を満たしていないことになります．しかし，直感的には，**図1**の設計は正しいように見えます．また，「信号aが2回連続して‘1’になると，その次の時刻に信号pが‘1’になる」というプロパティも正しいように思えます．これは，いったいどういうことなのでしょう？

図1の設計は，その状態遷移をたどってみても1ビット・カウンタとして正しいように見えますが，**図3**では状態S4が現れており，実際に反例（命題が正しくないことを示す例）を生成することもできます．実は，これはカウンタの動作の詳細をどう考えるかという問題であり，検証しようとしているプロパティがかならずしもそれに沿ったものではないことが原因です．

例えば，信号aが連続して3回‘1’になった場合，1ビット・カウンタはどのように動作するべきでしょうか？**図1**のカウンタでは，出力pを1回‘1’にするだけです．しかし，今考えているプロパティでは，信号aが2回連続して‘1’になると，出力pを次の時刻に‘1’にすることになっています．信号aが3回連続して‘1’になると，2回連続して‘1’になることが2回起こったことにもなるため，信号pが2回連続して‘1’にならないといけないことになっているのです．これが正しいプロパティかそうでないかは，1ビット・カウンタの動作の詳細をどう考えるかによります．

なお，これは，プロパティ検証の結果，「設計が正しくない」となった場合に生成される反例の一つであり，一般に，こうした検討が反例を解析することに相当します．

実際，1ビット・カウンタの仕様を「入力が2回‘1’になると，出力が1回‘1’になる」と日本語で述べた時点で，実は不明瞭なことを言っているのです．すなわち，入力が2回‘1’になって，その次の時刻に出力を‘1’にするときの入力の値もカウントするのかしないのか，明確ではありません．あるいは，上述の例のように入力が続けて3回‘1’になったとき，2回‘1’になったことが2回起きたと見るのか，そう見ないのかなどは，どちらとも解釈できます．通常のカウンタの概念では，3回続けて‘1’になっても，最初の2回でカウンタが出力を‘1’にしているので，入力が2回‘1’になったことが2回起きたとは考えないのが普通です．そのように考えると，検証しようとしていたプロパティがまちがっていたことになります．

● 自然言語やタイミング図の記述にはあいまいさがある

しかし，ここでよく考えてください．今の例では，「時相論理で仕様を書いたので，まちがったプロパティを記述した」とも言えますが，逆に言えば，「もとの日本語で表現した仕様があいまいで，厳密な仕様になっていない」とも考えられます．つまり，時相論理のようなあいまい性のない表現法を使って記述しないと，仕様を厳密に定義できないとも言えます．

また，日本語だけで仕様を記述しているからいけないのであって，タイミング・チャートなどの図形表現を併用して示せばよいと思われるかもしれません．しかし実際には，タイミング・チャートで表現しても，状況はそれほど改善されません．1ビット・カウンタの仕様として，単純に**図4**のようなタイミング・チャートで表現しても，もとの日本語以上の情報はありません．入力が3回続けて‘1’になった場合の動作については，何も触れられていないからです．

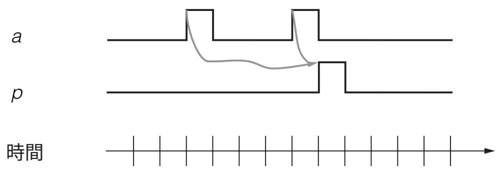


図4　1ビット・カウンタのタイミング・チャートの例

入力信号が2回になれば，出力信号が‘1’になるということは表現されているが，詳細な動作，例えば入力が3回続けて‘1’になった場合にどういう動作を行うべきかは示されていない．言い換えると，タイミング・チャートは実行例を表現しているにすぎないと言える．つまり，詳細な仕様とするには，すべての可能な場合についてタイミング・チャートを作成する必要がある．これは現実的とは言えない場合が多い．

タイミング・チャートによって，具体的にそのような場合の動作を示してやる必要があります．つまり，想定されるすべてのケースのタイミング・チャートを示さないと，不明確になったり，誤解を生じることになります．

そうであるとすると，結局，時相論理で仕様を記述しようとすることと必要な手間やわかりやすさは本質的に同じということになり，タイミング・チャートは図で表現しているがゆえに意味が明確でない部分もあるので，時相論理で記述したほうがよいとも言えます．また，記述した時相論理式は，前回の記事で示した方法によって状態遷移表現に展開できます．これをもとに，プロパティに対してシミュレーションを実行し，どのような動作が許されて，どのような動作が許されないのかを確認することもできます．このあたりの処理については，次回以降，仕様記述言語を説明するときに詳しく述べたいと思います．

実際には，仕様はタイミング・チャートとそれを説明する日本語記述によって表現されることが多く，そのような仕様を用いてハードウェアは設計され，実際に動作しています．どうしてそうなっているのかというと，「設計の常識」が暗黙のうちに仮定されているためです．タイミング・チャートには典型的な場合と例外の場合が示されるのが普通で，それ以外の場合については，「常識に従って動作する」といった無言の了解事項があるわけです．しかし，形式的に仕様を記述するということは，もれなく動作を定義する必要があるわけで，タイミング・チャートの意味を正確かつもれなく定義しないと，仕様記述としては使えないことになります．ただ，正確かつもれなくと言っても，定義する「人」によっていろいろと差がでてくるので，簡単にはいきません．

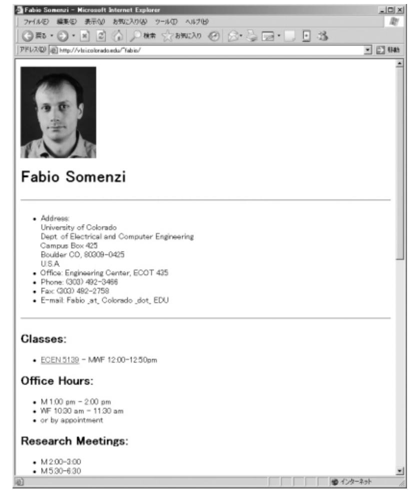


図5　http://vlsi.colorado.edu/~fabio/ のサイト

米国University of Colorado，BoulderのSomenzi教授のグループは，時相論理式を状態遷移表現に自動的に展開するツールを開発し，Web上で公開している．

● 時相論理式を状態遷移表現に展開するツールを公開

以上からもわかるように，時相論理で仕様を記述することは，論理で仕様を表現することになるので，意味が不明確な部分を「常識で補う」ということができません．完全な仕様を論理として表現しないと処理できないのです．このため，正しい仕様を時相論理で表現することに慣れる，あるいは訓練する必要があります．前回の説明からもわかるように，時相論理の動作は，その時相論理式を状態遷移表現に展開すると理解しやすくなります．実は，米国University of Colorado，BoulderのSomenzi教授のグループが，時相論理式を状態遷移表現に自動的に展開するツールを開発し，Web上で公開しています．これを利用して，いろいろと検討してみることにします．

まず，http://vlsi.colorado.edu/~fabio/ のサイト(**図5**)を開いて，そのページの中の「Wring」という部分を見つけてください．このWringというのが時相論理式を状態遷移表現に展開してくれるツールです．クリックするとツールのダウンロードが始まります．

このプログラムはPerlというインタープリタ型のプログラミング言語で書かれており，Perlの処理系があればどこでも動作します．もちろんLinuxではそのまま使えますし，Windowsの場合なら，Cygwinというツールを別途ダウンロードすることで使えるようになります(http://cygwin.com/から無償でダウンロード可能)．

ダウンロードしたWringを展開すると，ltl2aut.plという

```
---------------------------------------------------------------------
G(((p=1)*X(p=1))->X(X(q=1)))
---------------------------------------------------------------------
(FALSE)R(((p=0)+(X(p=0)))+(X(X(q=1))))
States
n2: {q=1,X((FALSE)R(((p=0)+(X(p=0)))+(X(X(q=1))))),X(X(q=1)),X(q=1)} label: {q=1}
n5: {p=0,q=1,X((FALSE)R(((p=0)+(X(p=0)))+(X(X(q=1)))))} label: {p=0,q=1}
n8: {p=0,X((FALSE)R(((p=0)+(X(p=0)))+(X(X(q=1)))))} label: {p=0}
n10: {X((FALSE)R(((p=0)+(X(p=0)))+(X(X(q=1))))),X(X(q=1)),X(q=1)} label: {}
n11: {X(p=0),X((FALSE)R(((p=0)+(X(p=0)))+(X(X(q=1)))))} label: {}
Arcs
   n2→{n2,n5}
   n5→{n11,n8}
-> n8→{n11,n8}
   n10→{n2,n5}
-> n11→{n10,n8}
Fair Sets
End
Stats: 5 states, 10 trans, 0 fair sets, 2 init states, weak
CPU time: 0.16 seconds
```

図6
G(((p=1)*X(p=1))->X(X(q=1)))の状態遷移表現
時相論理式を状態遷移表現に展開するツール「Wring」を利用して生成した．

ファイルがあると思います．これが時相論理式を状態遷移表現に展開するプログラムです．ltlとは“Linear Time Temporal Logic”のことを表しています．時相論理にはいくつかの種類があり，その中のltlを扱えるプログラムという意味です．

ちょっと実行してみましょう．これまでよく出てきた「信号pが2回続けて‘1’になると，信号qがその次の時刻で‘1’になる，ということがいつも起こる」というプロパティを例に説明します．この時相論理式は，Wringでは以下のように書きます．

```
G(((p=1)*X(p=1))->X(X(q=1)))
```

pやqが‘1’，‘0’になるということは等号で表現します．あとは，これまでと同じ表記法になっています．この式をファイルに格納し，そのファイル名がtest1.ltlであるとした場合，

```
ltl2aut.pl -ltl test1.ltl
```

を実行すると，時相論理式が状態遷移表現に展開されます．その結果が**図6**です．

図6の最後を見ると，状態が五つ生成され，それらの間に10の状態遷移があることがわかります．状態名はn2，n5，n8，n10，n11となっています．番号が飛んでいるのは，同じ処理を実行している状態を一つにまとめるなど，状態遷移表現の簡単化処理を行ったためです（いくつかの冗長な状態が削除された）．今回の話とは直接関係はありませんが，けっこう番号が飛んでいることから，最初は倍以上の状態が生成され，かなりのものが冗長であったことがわかります．読者のみなさんは，**図6**の結果を状態遷移図に書き換えて，その動作を確認してみてください……と言いたいところなのですが，このトレース情報はあまりわかりやすいとは言えません．

実は，このツールには**図6**のような表現ではなく，それをVerilog HDLとして出力する機能が付いています．ハードウェア設計者の方にはこちらのほうがわかりやすいと思います．次のコマンドを実行してください．

```
ltl2aut.pl -ltl -mon test1.v test1.ltl
```

この中で，-mon test1.vの部分でVerilog HDLによる記述をtest1.vに出力するように指定しています．実際のtest1.vの中身は**リスト1**のようになります．**リスト1**の中で，ND_n11_n8，ND_n10_n8，ND_n2_n5などについては，以下のような定義があります．

```
assign ND_n11_n8 = $ND(n11,n8);
assign ND_n10_n8 = $ND(n10,n8);
assign ND_n2_n5 = $ND(n2,n5);
```

$NDは，引き数のどちらかをランダムに選択する関数です．例えばND_n11_n8については，n11とn8のどちらか一方がランダムに選択されます．これは，与えられた時相論理式を表現するにはどちらが選ばれても正しいので，「どちらでもよい」ということを示すためにこのようになっています．実は，時相論理式を状態遷移表現に展開すると，次の状態が複数の状態のどれでもかまわないというnon-determinism（非決定性）が生じるのですが，$ND関数はそれをVerilog HDLで表現するためのくふうなのです．このように，時相論理式を自分で書いて，対応するVerilog HDLをプログラムに計算させ，それを調べるということを何度か実際にやってみると，時相論理でかなり自由に仕様

リスト1 `G(((p=1)*X(p=1))->X(X(q=1)))` **に対応するVerilog HDL記述**

```
typedef enum {n2,n5,n8,n10,n11,Trap} states;

module Buechi(clock,q,p,safe);
  input clock,q,p;
  output safe;
  states reg state;
  states wire ND_n11_n8;
  states wire ND_n10_n8;
  states wire ND_n2_n5;
  assign ND_n11_n8 = $ND(n11,n8);
  assign ND_n10_n8 = $ND(n10,n8);
  assign ND_n2_n5 = $ND(n2,n5);
  assign safe = (state != Trap);
  initial state = n8;
  always @ (posedge clock) begin
    case (state)
      Trap:
      state = Trap;
n5,n8:
      case (p)
      1'b0: state = ND_n11_n8;
      1'b1: state = n11;
      endcase
  n11:
      case (p)
      1'b0: state = ND_n10_n8;
      1'b1: state = n10;
      endcase
n2,n10:
      case ({p,q})
      2'b?0: state = Trap;
      2'b01: state = ND_n2_n5;
      2'b11: state = n2;
      endcase
    endcase
  end
endmodule
```

リスト2 `F(p=1)` **に対応するVerilog HDL記述**

```
typedef enum {n1,n2,n3} states;

module Buechi(clock,p,fair);
  input clock,p;
  output fair;
  states reg state;
  states wire ND_n1_n2;
  assign ND_n1_n2 = $ND(n1,n2);
  assign fair = (state == n3);
  initial state = n1;
  always @ (posedge clock) begin
    case (state)
     n2,n3:
     state = n3;
     n1:
     case (p)
     1'b0: state = n1;
     1'b1: state = ND_n1_n2;
     endcase
    endcase
  end
endmodule
```

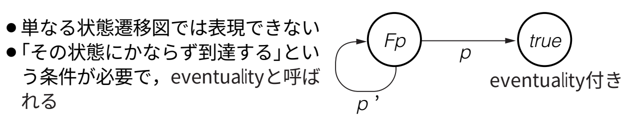


図7　*Fp*（いつかは*p*が成立する）ということ

***Fp*というのは，「今*p*が'1'になる」か，または，「次の時刻から見て，いつか*p*（つまり，*Fp*）が成り立つ」と言い換えることができるため，図のような状態遷移に変換できる．しかし，いつかは*p*が'1'になる必要があることを示すためには，いつかはtrueという状態に到達することを保証しないといけない．このようないつかは到達しないといけない状態のことを「eventualityのある状態」と呼ぶ．一般に，時相論理式を状態遷移に展開すると，必要に応じてeventualityのある状態が生成され，いつかはeventualityの付いている状態に到達する状態遷移列のみが許されるとみなされる．**

が書けるようになります．

●「いつか」を表現するeventuality

さて，今回は，もう一つだけ説明したいことがあります．それはeventualityと呼ばれる性質です．これは，時相論理式を状態遷移表現に展開したときに，状態を遷移していく過程でかならず通過しなければならない状態を指します．実は，時相論理演算子の一つである*F*演算子は「いつか起こる」ということを示しますが，この「いつか」を表現するためにeventualityの概念が必要になります．例えば，*Fp*というのは，「いつかは*p*が'1'になる」ということを示します．これを状態遷移表現にすると，**図7**のようになります．

この状態遷移表現で，いつも*p*の値が'0'だと（*p*'），trueというeventualityのある状態に決してたどり着かないことになり，そのようなシーケンスはもとの時相論理式を満たさないということになります．

さて，*Fp*を先ほどのツールで展開してみましょう．

```
F(p=1)
```

という記述をtest4.ltlというファイルに入れておき，

```
ltl2aut.pl -ltl -mon test4.v test4.ltl
```

を実行すると，**リスト2**のVerilog HDLファイルが生成されます．

リスト2の中には，以下のような定義があります．

```
assign fair = (state == n3);
```

状態`n3`に到達しないと`fair`という信号が'1'にならないことになり，これにより，状態`n3`にeventualityがあることがわかります．このようなeventuality付きの例題についてもいくつか自分で記述して，状態遷移表現に展開して確認してみてください．きっと，時相論理に対する理解が深まることでしょう．

さて，今回は，時相論理式を状態遷移表現に展開するツールを紹介しました．次回は，これを使って，時相論理の少し詳細な説明を行います．また，それに基づく仕様表現手法や仕様記述言語について紹介したいと思います．

ふじた・まさひろ
東京大学 大規模集積システム設計教育研究センター（VDEC）教授

<筆者プロフィール>
藤田昌宏．1985年，東京大学大学院 情報工学専門課程修了．工学博士．同年，富士通研究所入社．論理設計用CAD技術の研究開発に従事．1993年より米国富士通研究所へ出向，CAD研究グループを立ち上げる．2000年3月より東京大学大学院工学系研究科電子工学専攻教授，2004年4月より東京大学大規模集積システム設計教育センター（VDEC）教授．ディジタル・システム設計支援技術，特にシステム・レベル合成，論理合成，論理検証を中心に研究している．